浙江大学
ZHEJIANG UNIVERSITY

Computer Architecture Experiment

Topic 2. 5-stage CPU executing in pipeline & Pipelined CPU with stall

浙江大学计算机学院
陈文智
chenwz@zju.edu.cn

实验操作流程

- 阅读实验文档，理解处理器的各个功能模块组成和内部实现方式。
- 补全各个功能模块源代码中的空缺部分。
- 对处理器进行仿真，检验处理器的仿真结果是否符合要求。
- 综合工程并下载至开发板，在单步执行的过程中检查调试屏幕的输出，检验处理器的执行过程是否正确。

实验验收标准

- 仿真执行过程中，处理器的行为和内部控制信号均符合要求。
- 下载至开发板后的单步执行过程中，寄存器的变化过程和最终执行结果与测试程序相吻合。

Outline

- Experiment Purpose
- Experiment Task
- Basic Principle
- Operating Procedures
- Precaution
- Checkpoints

Experiment Purpose

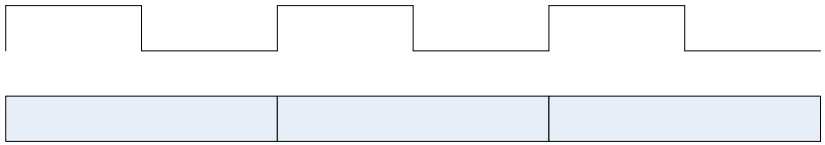
- Understand the principles of Pipelined CPU
- Understand the basic units of Pipelined CPU
- Understand the working flow of 5-stages
- Understand the principles of Pipelined CPU Stall
- Understand the principles of Data hazard
- Master the method of Pipelined CPU Stalls Detection and Stall the Pipeline.
- Master simulation of CPU Cores modules
- master methods of program verification of Pipelined CPU with Stall

Experiment Task

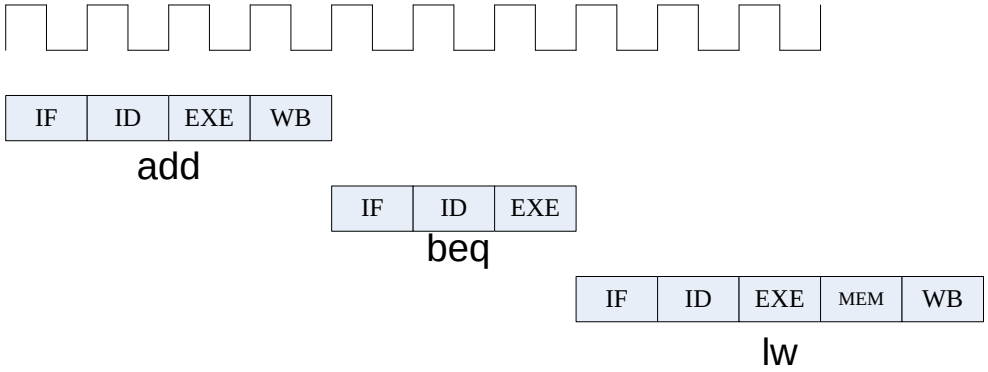
- **Design the CPU Controller**
- **Based on the CPU Controller, Design the Datapath of 5-stages Pipelined CPU**
  - 5 Stages
  - Register File
  - Memory (Instruction and Data)
  - other basic units
- **Design the Stall Part of Datapath of 5-stages Pipelined CPU**
- **Adding Condition Detection of Stall**
- **Simulating CPU Core module**
- **Verify the Pp. CPU with program and observe the execution of program**

Comparison of three CPUs' work

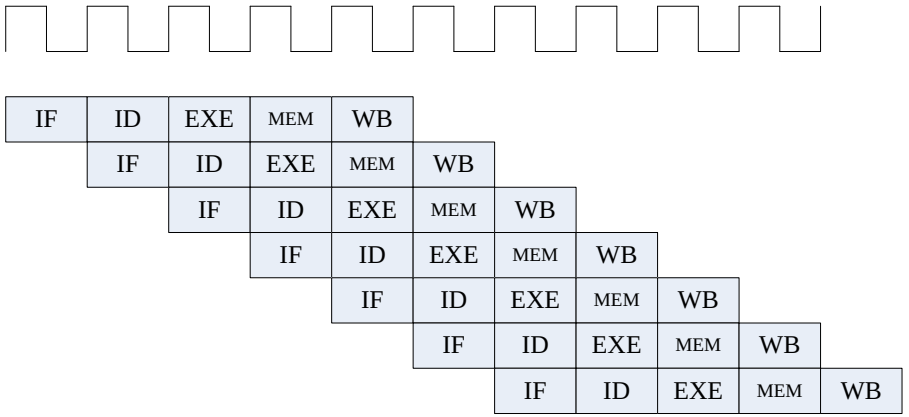
Simple-Cycle CPU



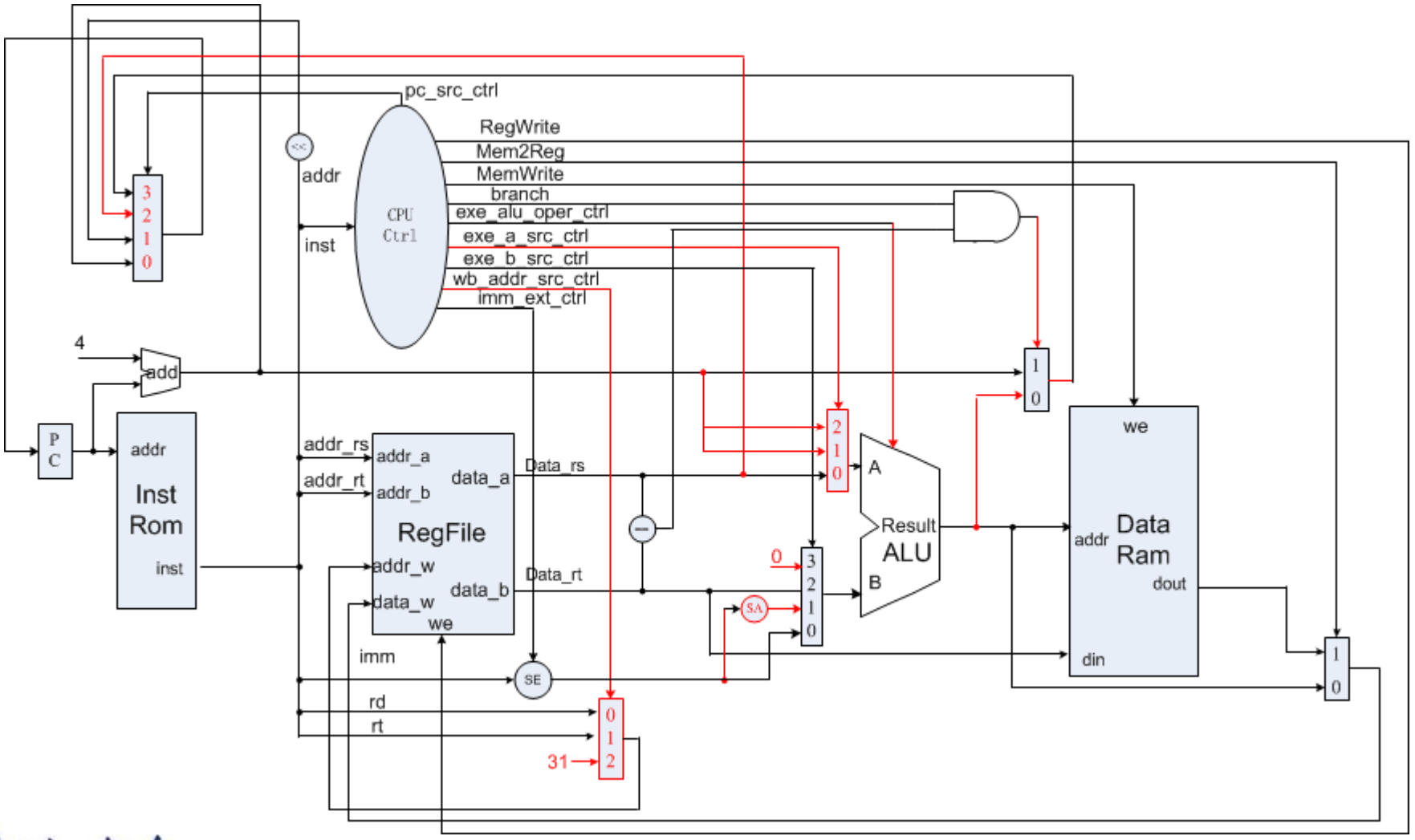
Multiple-Cycle CPU



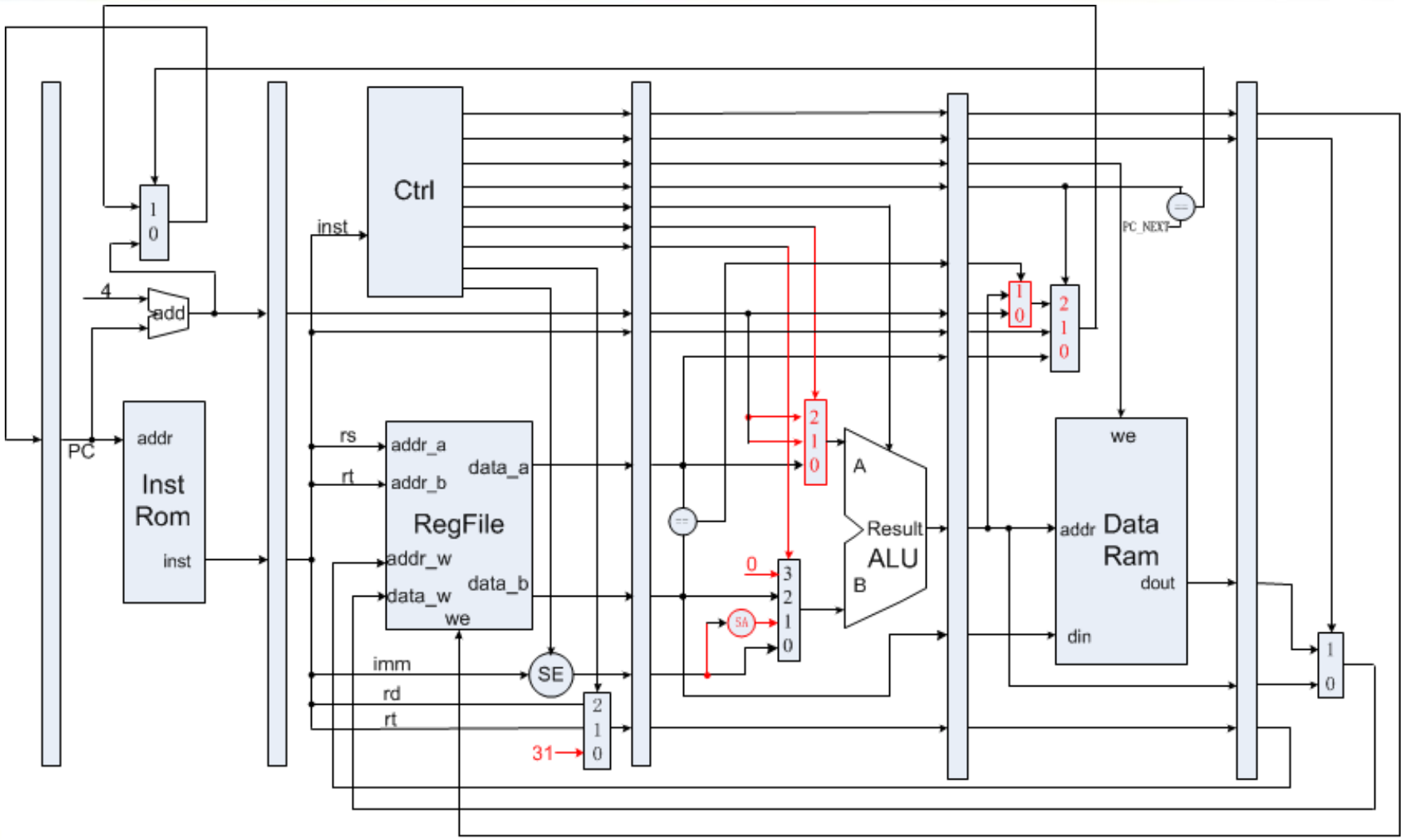
Pipelined CPU



The principle of Single-cycle CPU



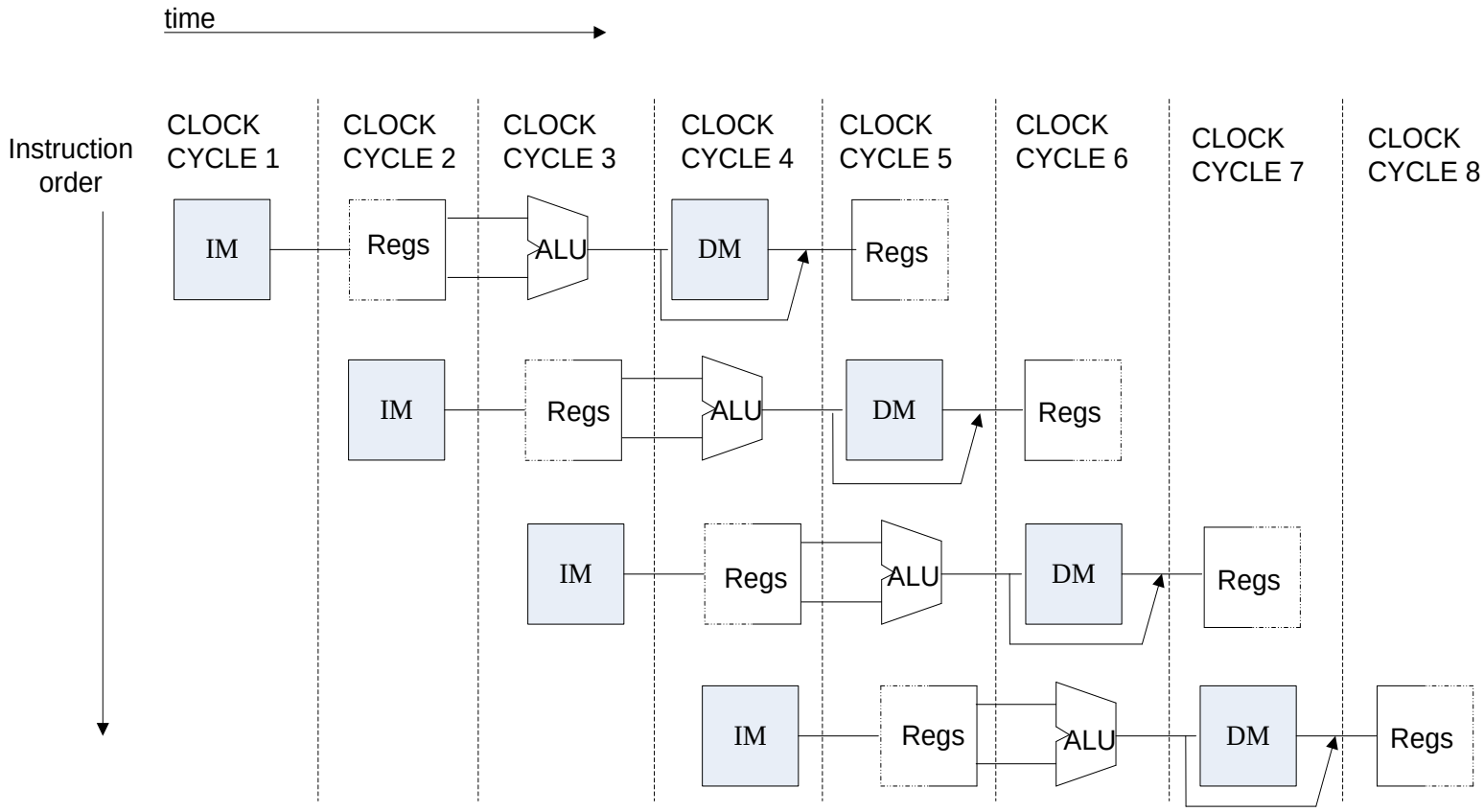
Datapath of 5-stages Pipelined CPU



Structural hazards —resource conflicts

- **Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.**
  - Memory conflicts
  - Register File conflicts
  - Other units conflicts

How to resolve Structural hazards



Memory

- **Instruction Memory**
  - Single Port Mem.
  - Read only, Width:32, Depth: 64
  - Falling Edge Triggered
- **Data Memory**
  - Single Port Mem.
  - Read and write, Width:32, Depth: 32
  - Falling Edge Triggered

Instruction Memory

```
parameter
        ADDR_WIDTH = 6;
reg [31:0] data [0:(1<<ADDR_WIDTH)-1];
initial  begin
        $readmemh("inst_mem.hex", data);
end
reg [31:0] out;
always @(negedge clk) begin
        out <= data[addr[ADDR_WIDTH-1:0]];
end
always @(*) begin
        if (addr[31:ADDR_WIDTH] != 0)
                dout = 32'h0;
        else
                dout = out;
end
```

Data Memory(1)

```
parameter
        ADDR_WIDTH = 5;

reg [31:0] data [0:(1<<ADDR_WIDTH)-1];

initial   begin
        $readmemh("data_mem.hex", data);
end

always @(negedge clk) begin
        if (we && addr[31:ADDR_WIDTH]==0)
                data[addr[ADDR_WIDTH-1:0]] <= din;
end
```

Data Memory(2)

```
reg [31:0] out;
always @(negedge clk) begin
        out <= data[addr[ADDR_WIDTH-1:0]];
end

always @(*) begin
        if (addr[31:ADDR_WIDTH] != 0)
                dout = 32'h0;
        else
                dout = out;
end
```

Register File

- **negative edge for write operation**
- **read operation at any time**
- **Double Bump**

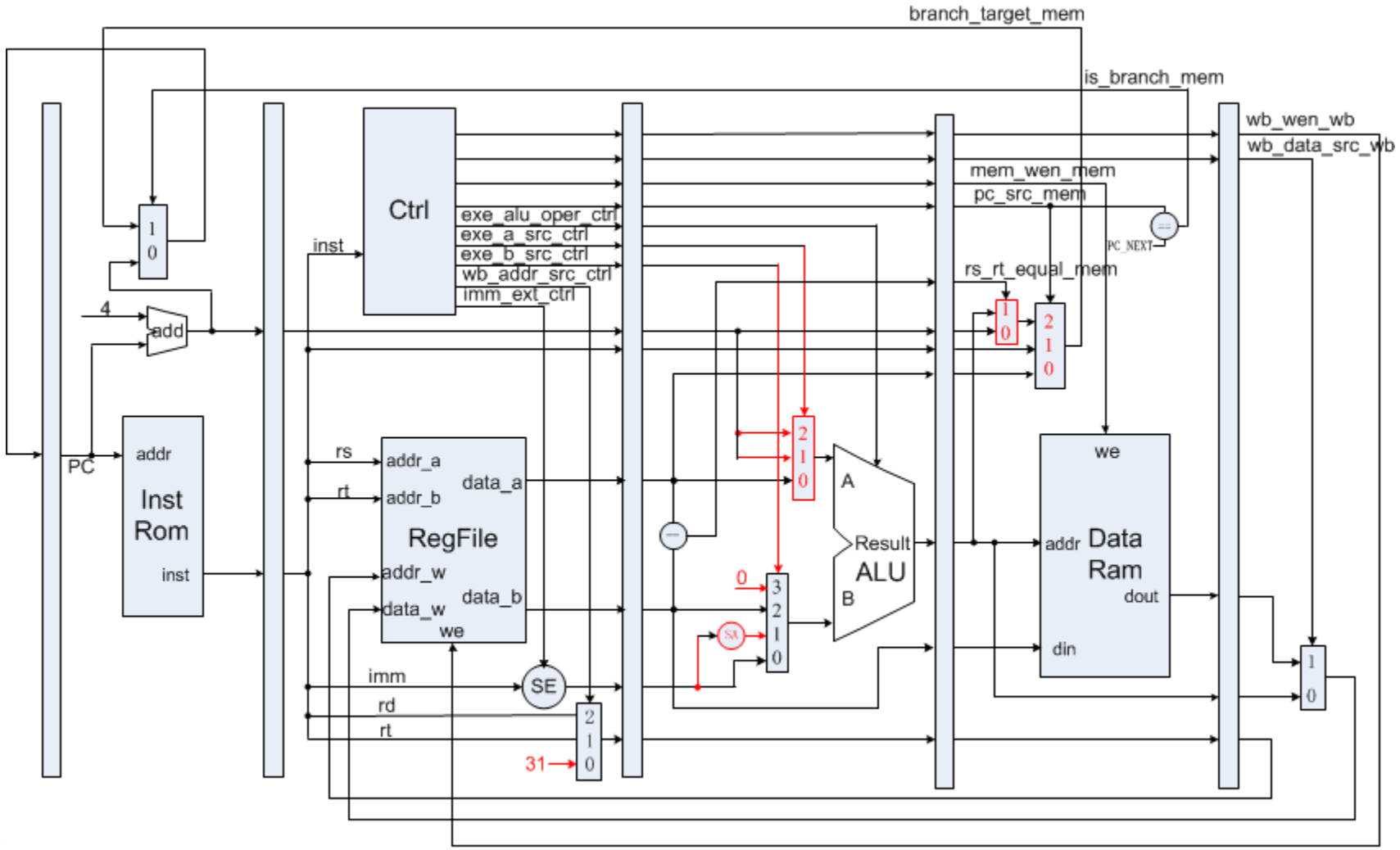
Register File

```
reg [31:0] regfile [1:31];  // $zero is always zero
// write
always @(negedge clk) begin
        if (en_w && addr_w != 0)
                regfile[addr_w] <= data_w;
end
// read
always @(*) begin
        data_a = addr_a == 0 ? 0 : regfile[addr_a];
        data_b = addr_b == 0 ? 0 : regfile[addr_b];
end
// debug
`ifdef DEBUG
always @(*) begin
        debug_data = debug_addr == 0 ? 0 : regfile[debug_addr];
end
`endif
```

Timing for Pipelined CPU

- Positive Edge: State Transfer
- Negative Edge
  - Instruction Mem. Read
  - Data Mem. Read/Write
  - Regfile Write
- Regfile Read: anytime

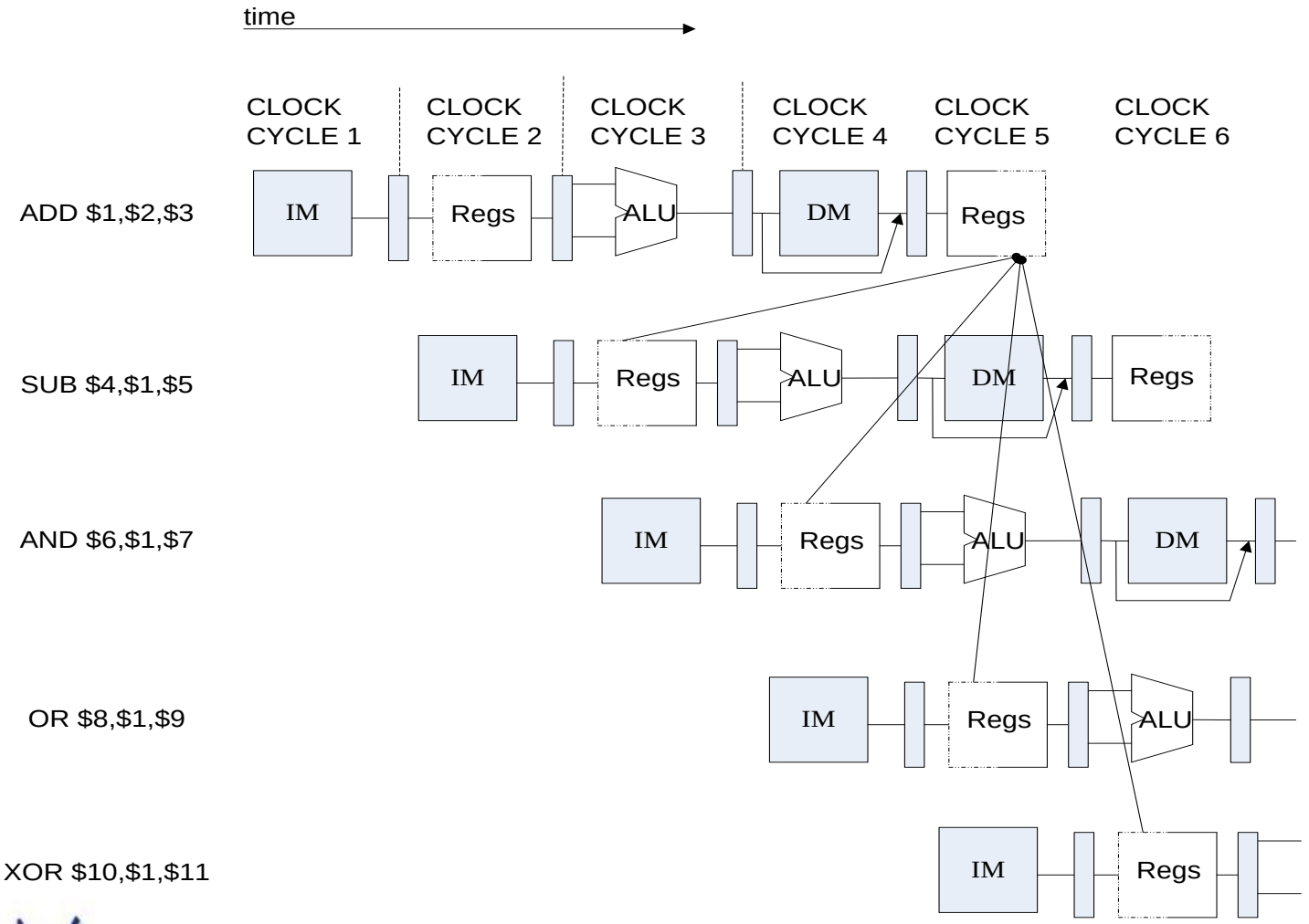
The principle of Pipelined CPU



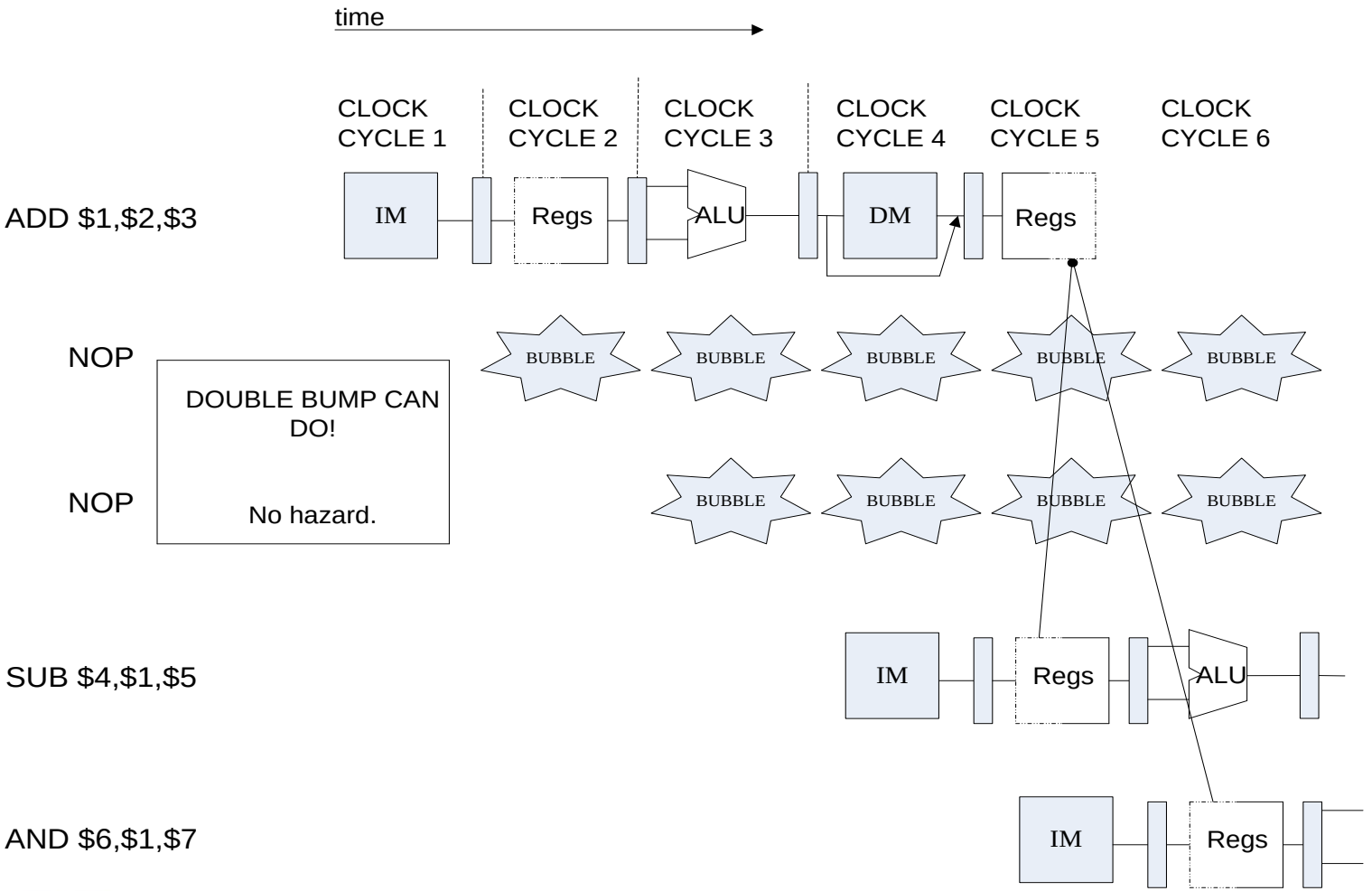
Data Hazard Definition

- **Data Hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.**
- **When inst. i executes before instr. j, there are data hazards:**
  - RAW, i.e. instr. j read a source before instr. i writes it.
  - WAW, i.e. instr. j write an operand before instr. i writes it.
  - WAR, i.e. instr. j write a destination before instr. i read it.

Instruction Demo



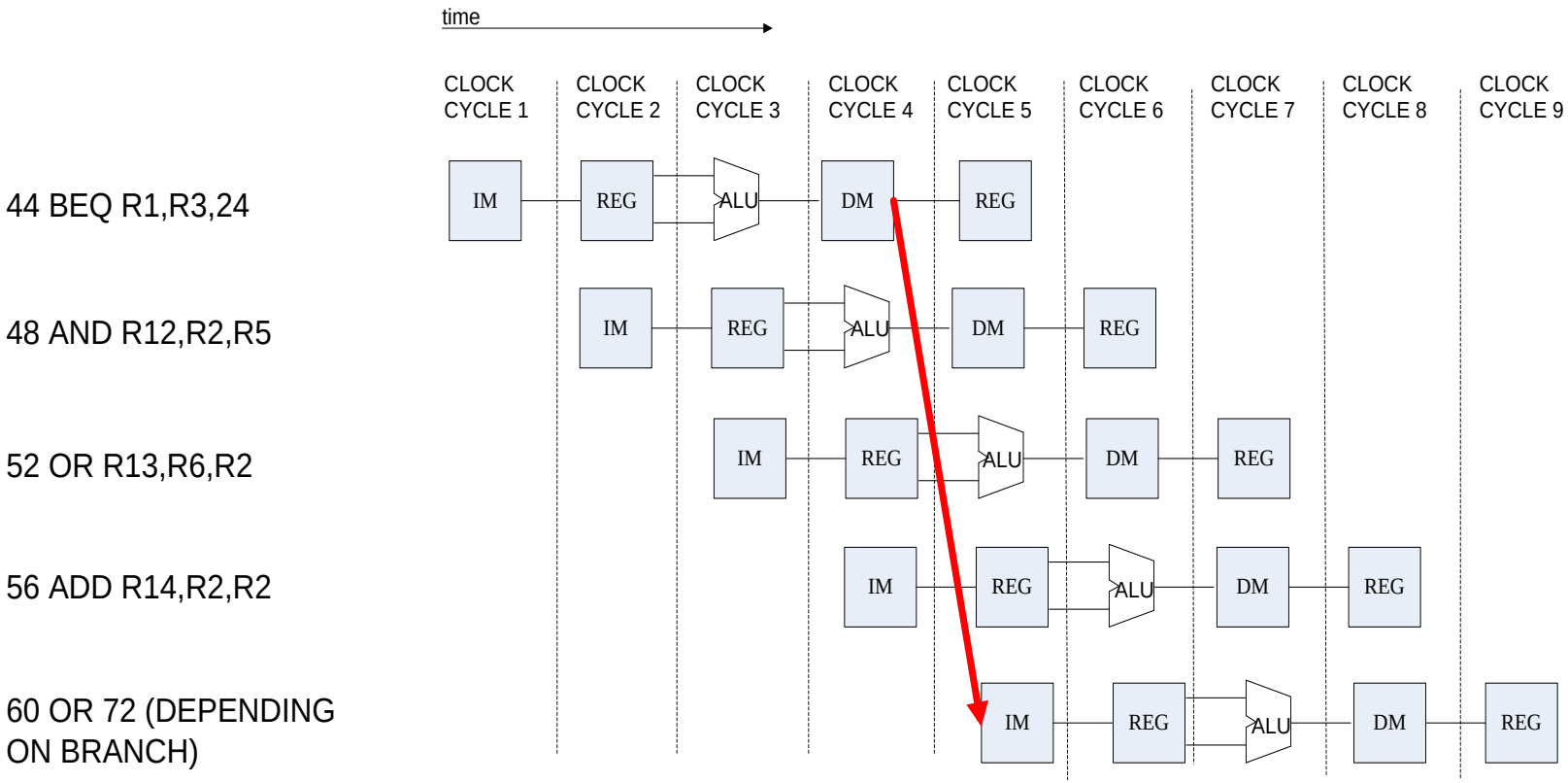
Data Hazard Causes Stalls



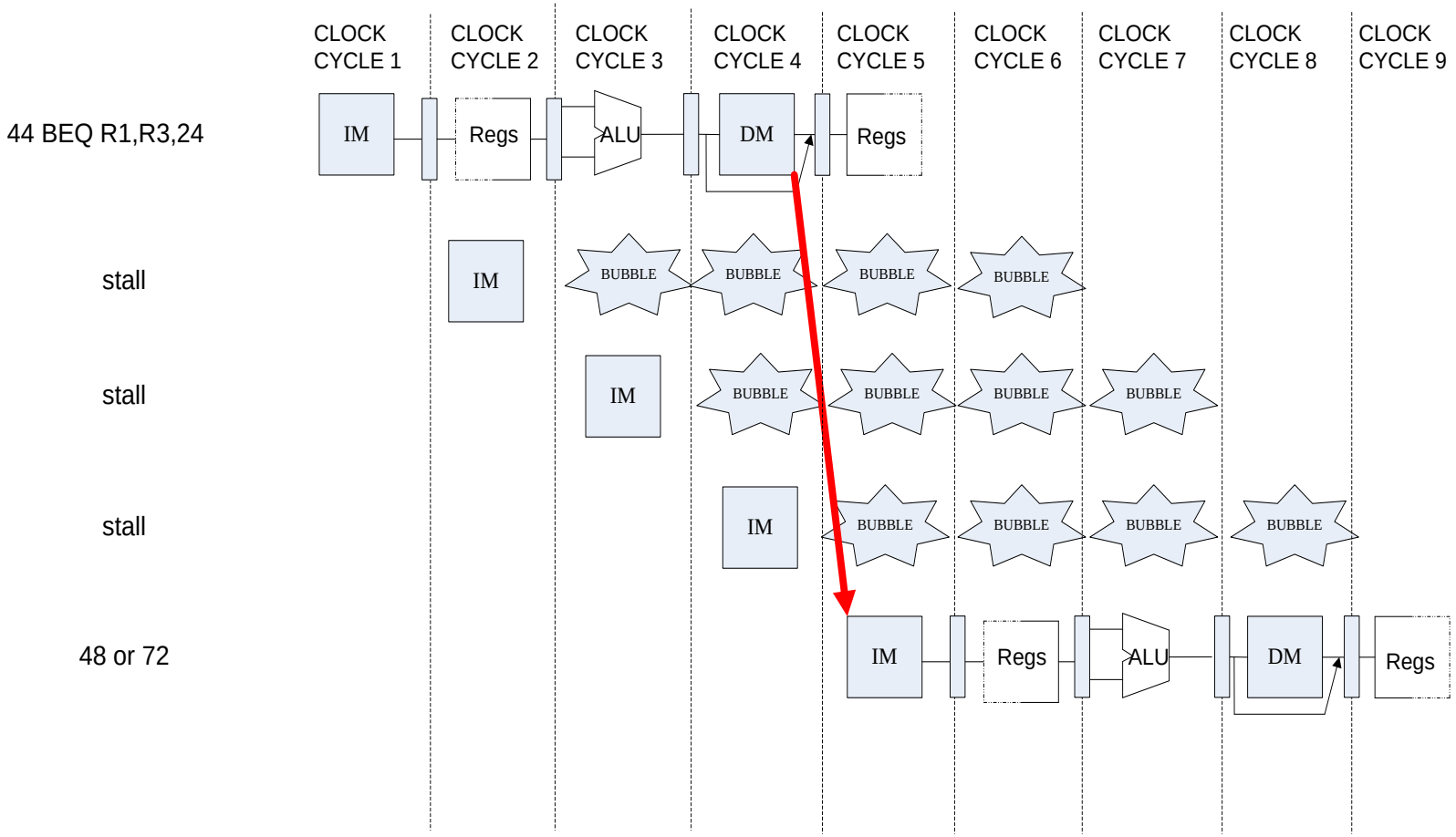
Control Hazard Definition

- **Control Hazards arise from the pipelining of branches and other instructions that change the PC.**

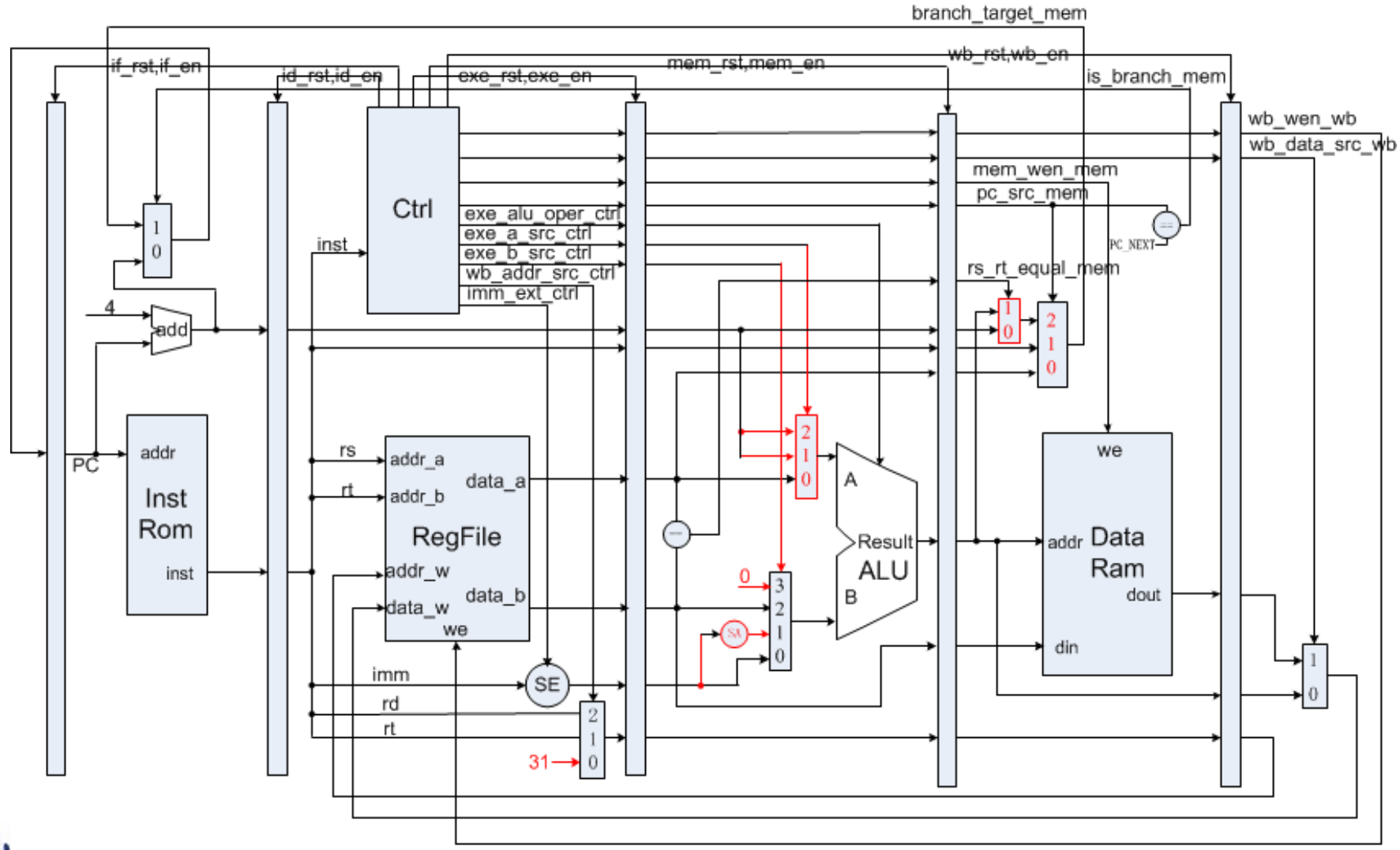
Execution result



Freeze method



How to Stall the Pipeline



In Which Conditions it Causes Stall(1)

- **AfromEx： //ID instr.rs=exe inst.rd and exe instr.writereg**

```
if (rs_used && addr_rs != 0) begin
        if (regw_addr_exe == addr_rs && wb_wen_exe) begin
                reg_stall = 1;
        end
```

- **Other expressions：**
- **BfromEx： id instr.rt=ex instr.rd and ex instr.writereg**
- **AfromMem： id instr.rs=mem instr.rd and mem instr.writereg**
- **BfromMem ： id instr.rt=mem instr.rd and mem instr = instr.writereg**

In Which Conditions it Causes Stall(2)

- **When ID Stage for BEQ INSTR.**

```
if (pc_src != PC_NEXT)
        branch_stall = 1;
```

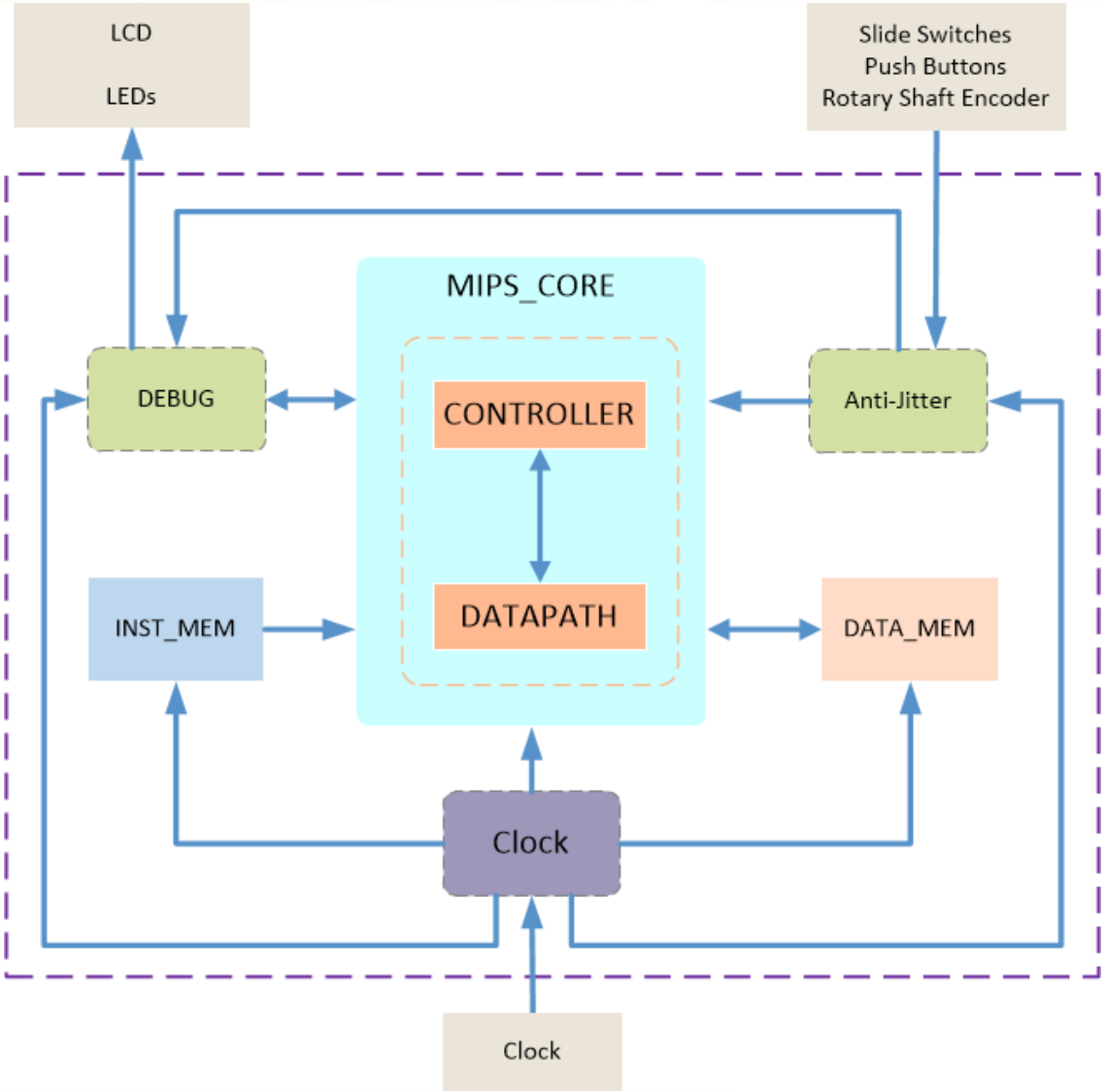
- **Other expressions：**
- **When EXE Stage for BEQ INSTR.**

- **When MEM Stage for BEQ INSTR.**

Project Introduction

- **Top Module**
- **MIPS Core**
- **Instruction Set**
- **Name**
- **Step Execution principle**
- **Observation Info**

Top Module (1)


LCD
LEDs
Slide Switches
Push Buttons
Rotary Shaft Encoder
MIPS_CORE
DEBUG
CONTROLLER
Anti-Jitter
INST_MEM
DATAPATH
DATA_MEM
Clock
Clock

Top Module (2)

- **anti-jitter**
- **clk_gen**
- **debug**
- **mips_core**
- **inst_rom**
- **data_ram**

MIPS Core - CPU Controller (1)

- **CPU Controller works in ID Stage**

- **Set signals according to different instruction**

MIPS Core - CPU Controller (1)

No.	singals	comment
1	inst	Input for id instruction
2	is_branch_exe	pc src in exe is PC_NEXT or not
3	regw_addr_exe	address source to write data back to registers
4	wb_wen_exe	memory write enable signal
5	is_branch_mem	pc src in mem is PC_NEXT or not
6	regw_addr_mem	address source to write data back to registers
7	wb_wen_mem	memory write enable signal
8	regw_addr_wb	data source of data being written back to registers
9	wb_wen_wb	register write enable signal
10	pc_src	pc src: PC_NEXT, PC_JUMP, PC_JR, PC_BEQ

MIPS Core - CPU Controller (2)

No.	singals	comment
11	imm_ext	If immediate is extended
12	exe_a_src	Control signal of A operand for ALU
13	exe_b_src	Control signal of B operand for ALU
14	exe_alu_oper	ALU operations
15	mem_ren	Mem read signal
16	mem_wen	Mem write signal
17	wb_addr_src	Control signal of write back addr
18	wb_data_src	Control signal of write back data
19	wb_wen	Control signal of write back enable

MIPS Core - CPU Controller (3)

No.	singals	comment
21	if_rst/if_en	Output for if stage controll
22	if_valid	Input for if stage stage
23	id_rst/id_en	Output for id stage controll
24	id_valid	Input for id stage stage
25	exe_rst/exe_en	Output for exe stage controll
26	exe_valid	Input for exe stage stage
27	mem_rst/mem_en	Output for mem stage controll
28	mem_valid	Input for mem stage stage
29	wb_rst/wb_en	Output for wb stage controll
30	wb_valid	Input for wb stage stage

MIPS Core - Datapath

- **Debug info output**
- **If subModule**
- **Id subModule**
  - Register file
  - Reg stall logic control
- **Exe subModule**
  - ALU
- **Mem subModule**
- **Wb subModule**

16 MIPS Instructions

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations	
R-type	op	rs	rt	rd	sa	func		
add	000000	rs	rt	rd	00000	100000	rd = rs + rt; with overflow	PC+=4
sub		rs	rt	rd	00000	100010	rd = rs - rt; with overflow	PC+=4
and		rs	rt	rd	00000	100100	rd = rs & rt;	PC+=4
or		rs	rt	rd	00000	100101	rd = rs | rt;	PC+=4
sll		00000	rt	rd	sa	000000	rd = rt << sa;	PC+=4
srl		00000	rt	rd	sa	000010	rd = rt >> sa (logical);	PC+=4
slt		rs	rt	rd	00000	101010	if(rs < rt)rd = 1; else rd = 0; <(signed)	PC+=4
jr		rs	00000	00000	00000	001000		PC=rs
I-type	op	rs	rt	immediate				
addi	001000	rs	rt	imm			rt = rs + (sign_extend)imm; with overflow	PC+=4
andi	001100	rs	rt	imm			rt = rs & (zero_extend)imm;	PC+=4
ori	001101	rs	rt	imm			rt = rs | (zero_extend)imm;	PC+=4
lw	100011	rs	rt	imm			rt = memory[rs + (sign_extend)imm];	PC+=4
sw	101011	rs	rt	imm			memory[rs + (sign_extend)imm] <-- rt;	PC+=4
beq	000100	rs	rt	imm			if (rs == rt) PC+=4 + (sign_extend)imm <<2;	PC+=4
J-type	op	address						
j	000010	address					PC = (PC+4)[31..28],address<<2	
jal	000011	address					PC = (PC+4)[31..28],address<<2 ; $31 = PC+4	

Name Specification

- **Signal of**
  - Stage: if, id, exe, mem, wb
  - Module: ctrl (id)

- **Signal name**
  - xxx_: works in xxx stage
  - _xxx: generated from xxx stage

Step Execution Principle

- **Stall all the stage.**
- **When step button is pushed, unlock the stall controllers of all the stages.**

- **Controller:**

```
else if ((debug_en) && ~(~debug_step_prev && debug_step)) begin
        if_en = 0;
        id_en = 0;
        exe_en = 0;
        mem_en = 0;
        wb_en = 0;
end
```

Reg Stall & Branch Stall

```
// this stall indicate that ID is waiting for previous  instruction, insert one NOP
between ID and EXE.
        else if (reg_stall) begin
                if_en = 0;
                id_en = 0;
                exe_rst = 1;
        end
//if  branch_stall is on, insert NOP in ID.
        else if (branch_stall) begin
                id_rst = 1;
        end
```

Observation Info - Input

- **South Button: Step execution**
- **North Button: Reset**
- **Rotary Shaft Encoder: Index Control**
- **Swtich[0]**
  - 0 for register display
  - 1 for data signal display: IF-ADDR/ IF-INST/ ID-ADDR/ ID-INST/ EX-ADDR/ EX-INST/ MM-ADDR/ MM-INST/ EMOPER(mem_ren+ mem_wen)/ MEMADDR/ MEMDATR/ MEMDATW/……
- **Swtich[3]:**
  - 0 for run
  - 1 for step execution

Observation Info - Output

- LCD
  - First line: Hello, world (reserved for Cache display)
  - Second line: Key + value
- LED
  - display for {buttons , switches}’s states

Program for verification (1)

```
00000824 main:    and $1, $0, $0      # address of data[0]
34240050          ori $4, $1, 80      # address of data[0]
20050004 call:    addi $5, $0, 4      # counter
0c00000a          jal sum             # call function
ac820000 return:  sw $2, 0($4)        # store result
8c890000          lw $9, 0($4)        # check sw
ac890004          sw $9, 4($4)        # store result again
01244022          sub $8, $9, $4      # sub: $8 <- $9 - $4
08000008 finish:  j finish            # dead loop
00000000          nop                 # done
```

Program for verification (2)

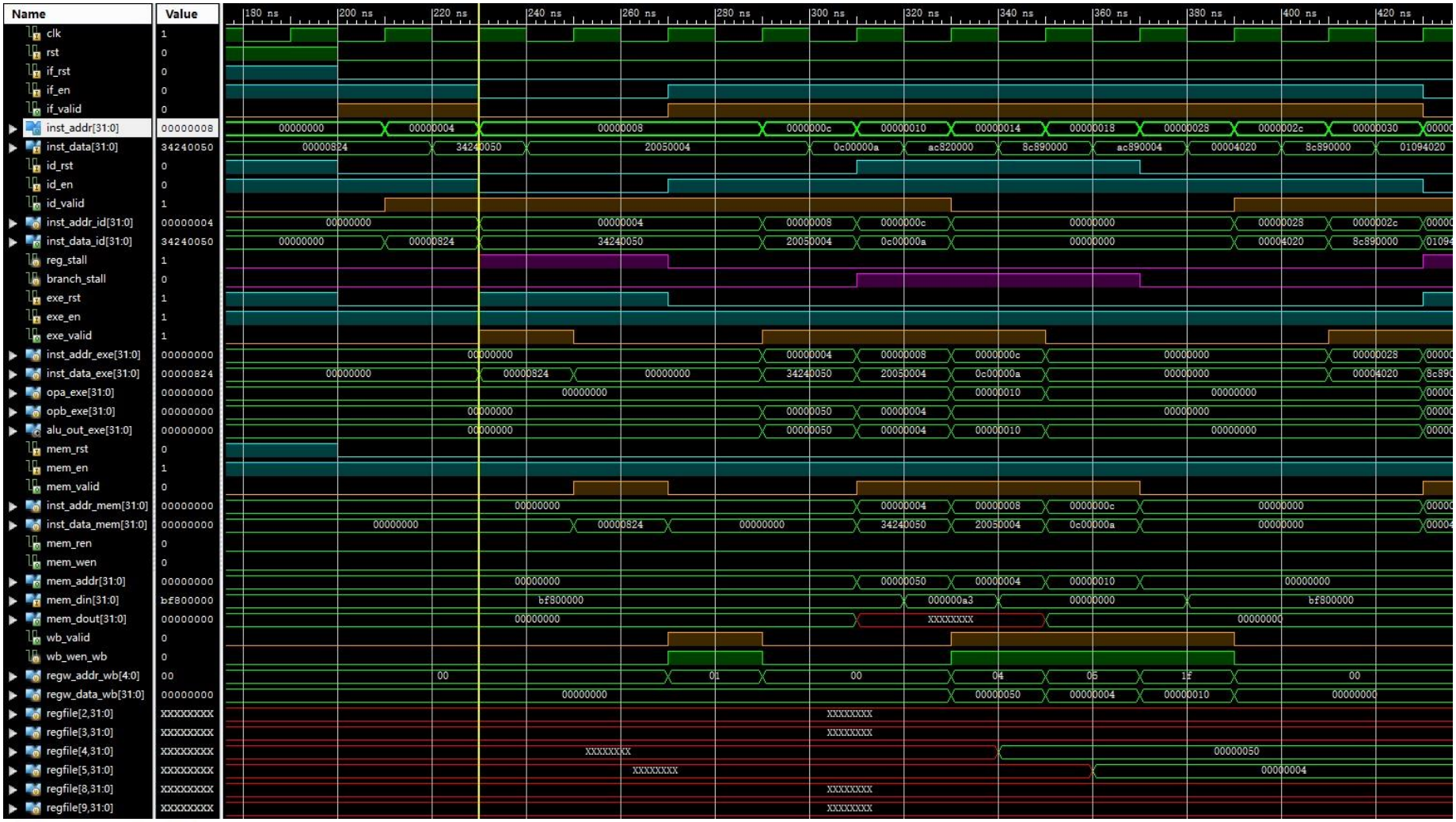
```
00004020 sum:   add $8, $0, $0      # sum function entry
8c890000 loop:  lw $9, 0($4)        # load data
01094020        add $8, $8, $9      # sum
20a5ffff        addi $5, $5, -1     # counter - 1
20840004        addi $4, $4, 4      # address + 4
0005182a        slt $3, $0, $5      # finish?
1460fffa        bne $3, $0, loop    # finish?
01001025        or $2, $8, $0       # move result to $v0
03e00008        jr $ra              # return
00000000        nop                 # done
```

Checkpoints

- **CP 1: Waveform Simulation of 5-stage Pipelined CPU**

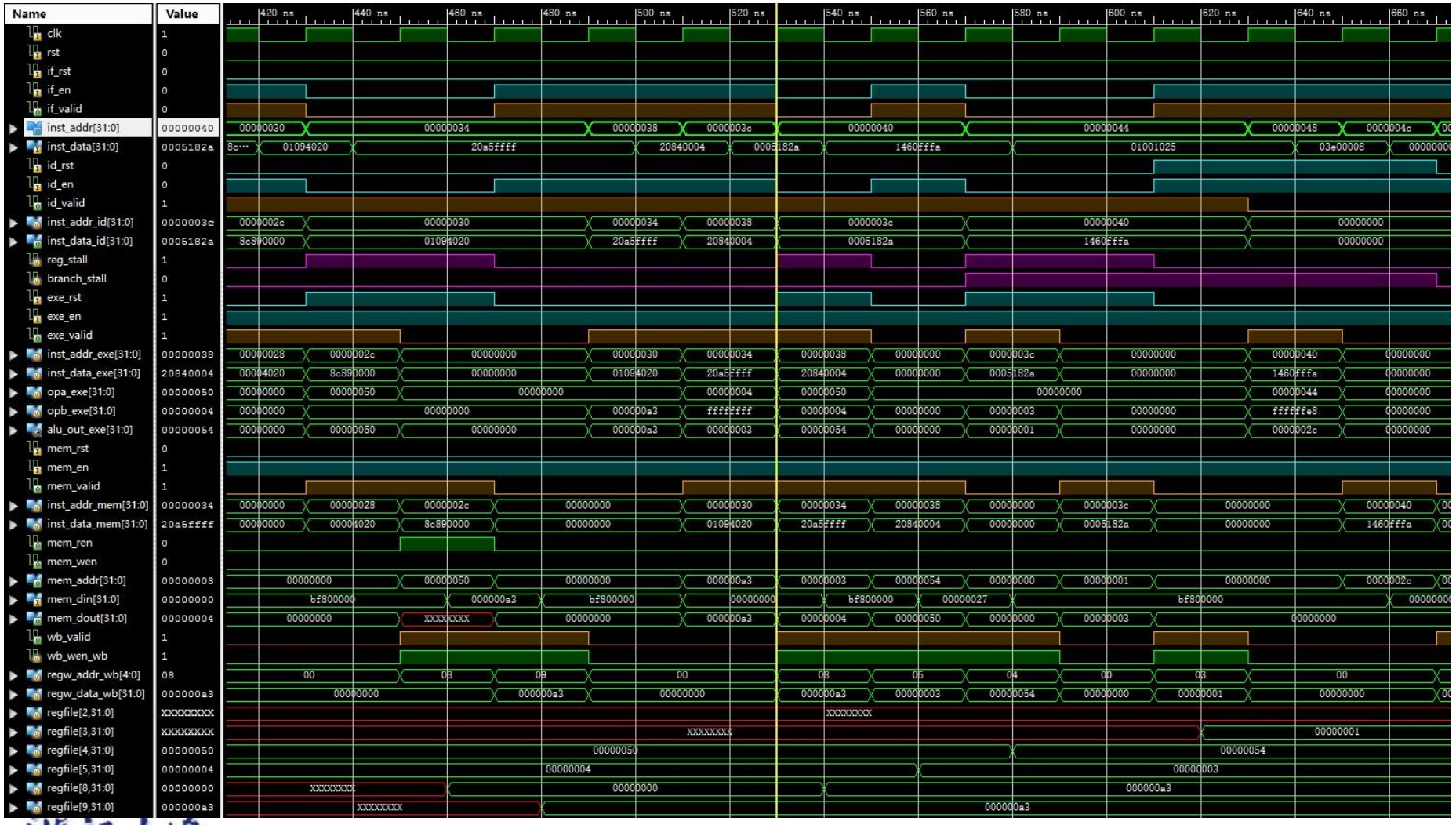
- **CP 2: FPGA Implementation of 5-stage Pipelined CPU with the verification program**

Simulation - 1

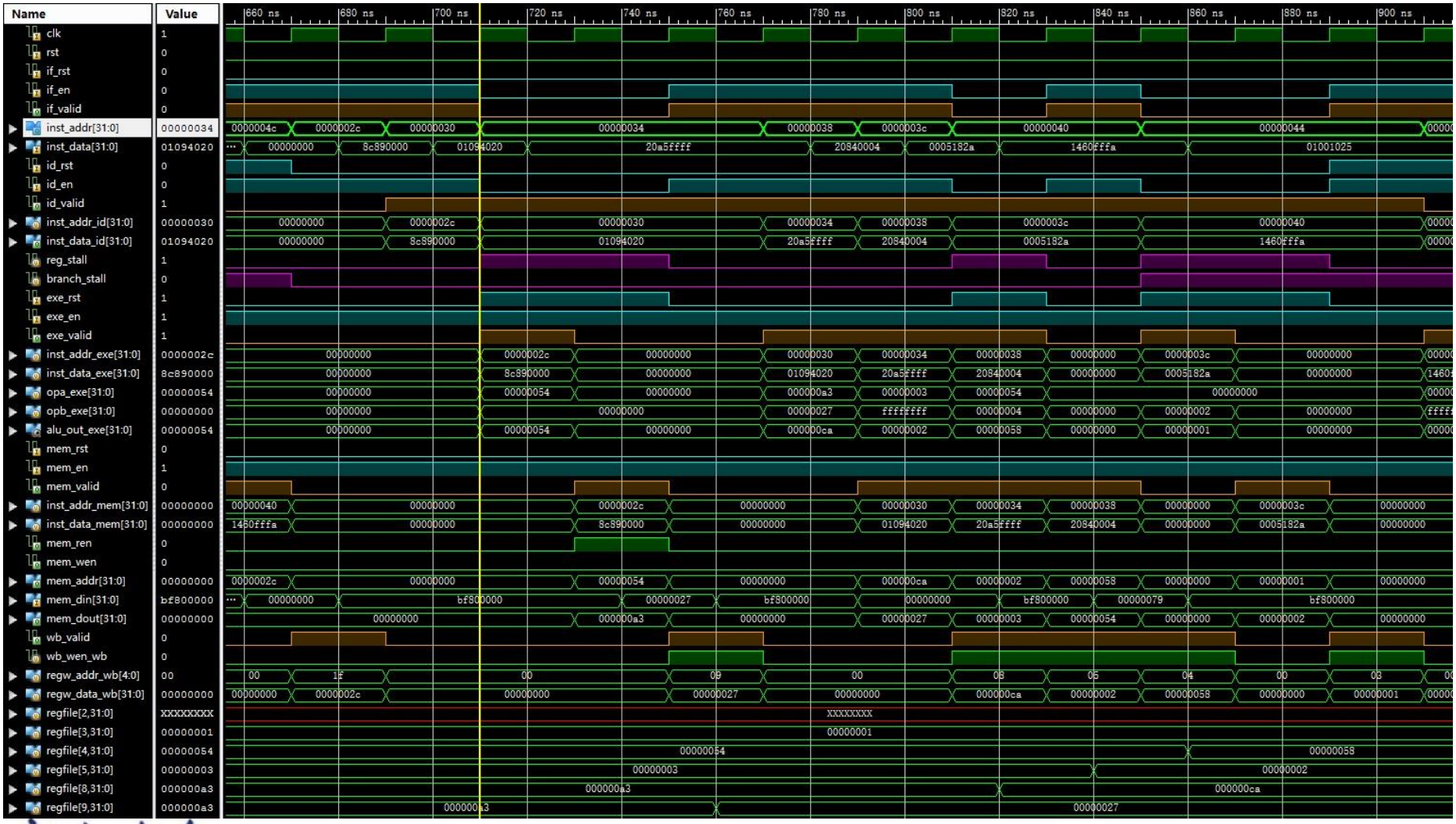


浙江大学 ZheJiang University

Simulation - 2

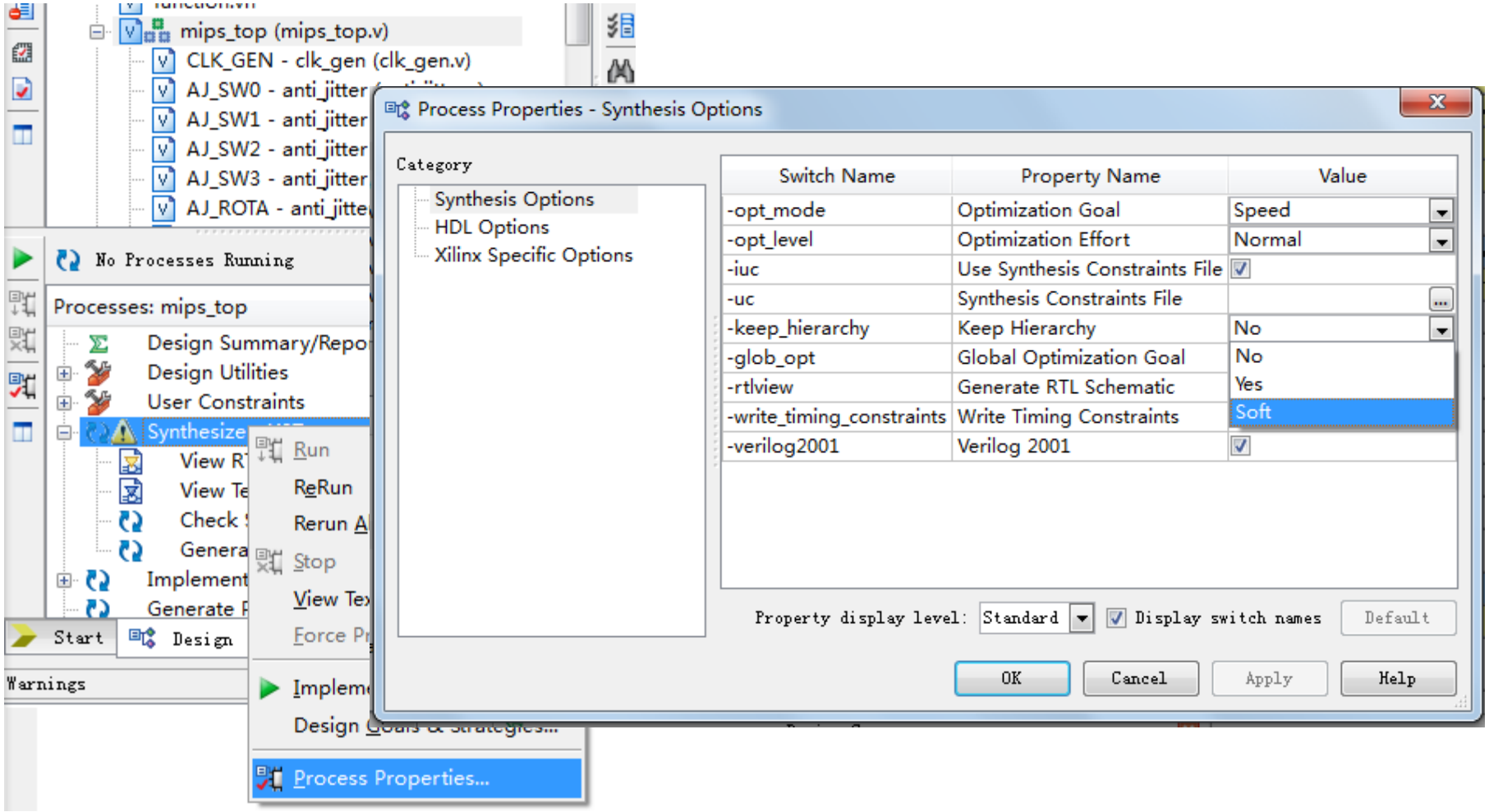


Simulation - 3



浙江大学 ZheJiang University

Settings


mips_top (mips_top.v)
CLK_GEN - clk_gen (clk_gen.v)
AJ_SW0 - anti_jitter
AJ_SW1 - anti_jitter
AJ_SW2 - anti_jitter
AJ_SW3 - anti_jitter
AJ_ROTA - anti_jitte
No Processes Running
Processes: mips_top
Design Summary/Repo
Design Utilities
User Constraints
Synthesize
View R
View Te
Check
Genera
Implement
Generate P
Start
Design
Warnings
Run
ReRun
Rerun A
Stop
View Tex
Force Pr
Impleme
Design Goals & Strategies...
Process Properties...
Process Properties - Synthesis Options
Category
Synthesis Options
HDL Options
Xilinx Specific Options
Switch Name
Property Name
Value
-opt_mode
Optimization Goal
Speed
-opt_level
Optimization Effort
Normal
-iuc
Use Synthesis Constraints File
-uc
Synthesis Constraints File
-keep_hierarchy
Keep Hierarchy
No
-glob_opt
Global Optimization Goal
-rtlview
Generate RTL Schematic
-write_timing_constraints
Write Timing Constraints
-verilog2001
Verilog 2001
No
Yes
Soft
Property display level:
Standard
Display switch names
Default
OK
Cancel
Apply
Help

Thanks!

浙江大学
ZheJiang University